ASIC Sign-Off in VHDL

Oz Levia and Fred Abramson

Synopsys Inc.

700 East Middlefield Road

Mountain View, CA 94043-4033

Abstract: Since its inception, VHDL has been promoted as a language for multi-level design. Proponents of the language have advocated its use partly on the grounds that it could accommodate design needs from high level requirements through RTL synthesis to gates, and gate level simulation. In recent years the language has enjoyed great success in satisfying design needs at high and intermediate levels of abstraction. VHDL is used extensively for simulation of behavioral and RTL descriptions. The language is also used for synthesis and optimization. For simulation at the gate level however, VHDL has not been as successful. Sign-off simulation, in particular, has traditionally been done using other languages. In this paper we present an approach to sign-off simulation of ASICs in VHDL.

1. Introduction

In ASIC design, sign-off simulation has a special meaning. In a sense, it is a contract between the designer and the silicon vendor. Since the cost of fabrication is so high and time consuming[1], a vendor must have a way of insuring the correctness of the design before fabrication begins. Specifically, it must be shown that the timing relationships that result from the chip layout do not violate the permitted limits. To accomplish this, the vendor specifies its process characteristics in a library of parts and requires that a design be simulated on a Sign-off (or **Golden**) Simulator before the design is allowed to be fabricated. The vendor, in turn, promises to deliver working silicon if the design does pass the simulation. As a result, the economic implications are tremendous, and vendors are quite selective in certifying a simulator as sign-off quality. In the past, VHDL simulators did not receive sign-off certification for a variety of reasons, and many vendors were relying on in-house simulators or on other languages (Verilog). From the designer's point of view, however, the most appropriate simulator for sign-off is the simulator that is used throughout the design process (whatever it may be), since using that simulator eliminates the need to switch tools at the end of the process. A need exists for VHDL and VHDL simulators to be sign-off "strong". In this paper we present an approach to making VHDL, and VHDL simulators, sign-off quality.

We begin by defining and describing the requirements for sign-off. We explore why VHDL has not satisfied such requirements in the past. Then we look at how our approach builds sign-off capability on top of VHDL, and -- item by item -- how our approach satisfies each one of the requirements. We offer some examples showing how VHDL is used to specify functionality and timing to the standards required for sign-off.

2. Sign-Off Requirements

Sign-off simulation is a verification step in the design process. Once the design has been simulated correctly using a Sign-off Simulator, the certifying ASIC vendor guarantees production of

1. But also because fixing problems in the design after fabrication (re-design) is very costly.

working silicon. To be certified as Sign-off, a simulator must have certain capabilities and functions. We distinguish three separate categories of requirements, each dealing with a certain aspect of the Sign-off process:

2.1. Timing and Functional Accuracy Requirement

In general, a logic simulator must be able to correctly predict circuit behavior. This is naturally true for Sign-off simulation as well. The simulator must be able to manipulate gate level objects and must also be able to evaluate them correctly. In addition to verifying that the circuit functions properly on the expected inputs, the simulator must also simulate *dysfunctional* inputs correctly. Together, these requirements ensure that the circuit produces the required output values just in situations the simulator predicts that it will.

Functional simulation is but part of the accuracy requirement. Timing is the other part. In fact, a major goal of the sign-off simulation is to verify that correct timing relationships exist in the design. In ASIC libraries, timing (or gate delay) is specified in terms of delays from certain inputs to certain outputs. This relationship model is called the *pin-to-pin delay model*, and a sign-off simulator must be able to make use of such a model in simulation. Using this delay model, a path is created from each input to each output, and several delay values are assigned to the path, one for each different kind of output transition.

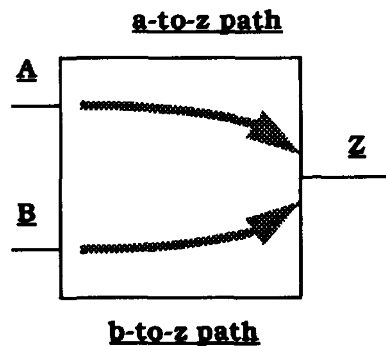

Figure 1. Pin-to-Pin delay Model

This delay model is simple and easy to use, but some ASIC cells require timing relationships that are more complex. For example, the delay value that is selected may depend on the state (or value) of other inputs to the cell. This delay model is called *State Dependent Delay* since the delay is selected using an expression which represents a particular state of the cell.

In addition to selecting the appropriate delay value for a cell, the simulator must also perform various checks to validate timing relations that occur in the model. For example, the simulator must validate the *setup* and *hold* relationships between the data pins and the clock pin, and the clock rise and fall periods.

2.2. Performance Requirement

Sign-off simulation is a final verification step. Often the whole design is simulated using a substantial set of vectors. It is not uncommon to simulate a 50K-150K gate design (or even a larger design) with several hundred thousand vectors. In the near future, 1M gate designs will become the norm. The realities of such design size dictate that a Sign-off simulator must be fast and allow for high capacity. While specific speed and capacity requirements are meaningful only as a comparison and not as absolute measures, note that for a 1M gate design with 1M test vectors

to fit on a next generation work station (256 Meg of RAM), the simulator must use no more then 256 bytes per *ASIC gate*. And if simulation is to take less than a day (24 hours), the simulator must be able to provide performance at the rate of 10-12 cycles (vectors) per second[1].

2.3. Miscellaneous Requirements

From the ASIC vendor's point of view, a Sign-off simulator should produce behavior that is as close as possible to the actual silicon process. This has significant implications on requirements related to simultaneous changes in the circuit (more specifically -- *near* simultaneous changes of *inputs* to a specific ASIC cell). Accordingly, a Sign-off simulator is expected to check for hazards (glitches or spikes) in specific situations.

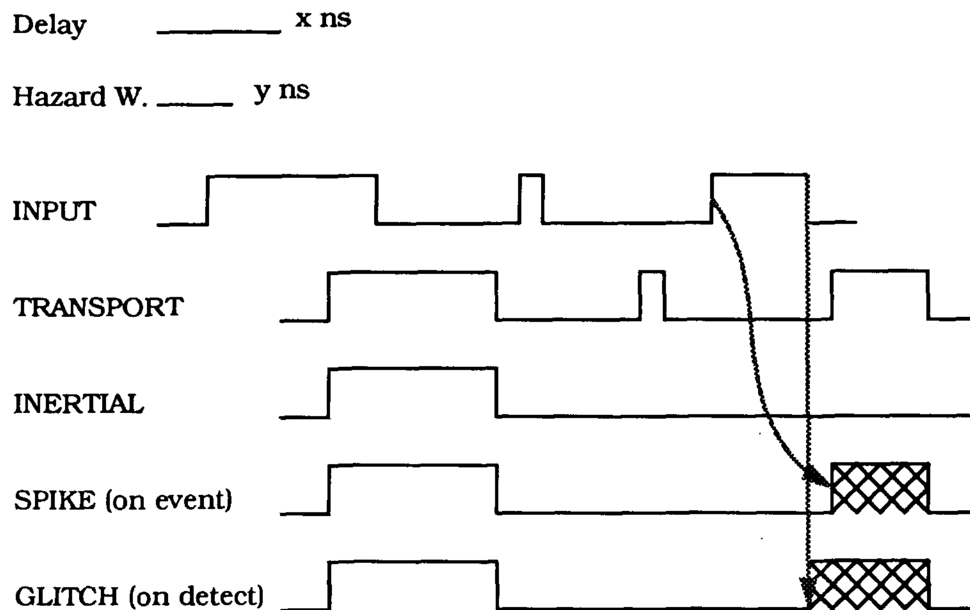

Figure 2. Hazard and Glitch Detection

Once a hazard is detected, the simulator should *respond* in a certain fashion.

Other situations also require detection and action, for example: hold, setup, and clock period. But the reaction to violation detection requires some "intelligence". The simulator should be able to detect when a violation is not important (e.g. a hold violation if reset is active). More examples of violations that require detection are: Bus contention (more than one driver) and "set" and "reset" hold and set relationship.

Once simulation is complete (or while it is in progress) the ASIC vendor requires additional information about the design. For example, an ASIC vendor may require a toggle count complete with node coverage. And of course, an appropriate tester format (to match the one used by the vendor) is also required.

1. This is just an approximation. It does NOT a representation of any actual tool.

3. VHDL and ASIC Simulation - Problem Description

Using VHDL as a description language for sign-off libraries and simulation makes a lot of sense. For one, VHDL is a standard language. This means that modeling efforts can be leveraged, and that libraries can be used on more platforms. VHDL is also a rich language that can be used at many levels of the design cycle, relieving the designer from the need to convert the design for the last verification step. To understand why VHDL has not been successful in penetrating this market segment one need only examine the language itself. Specifically, we believe VHDL is too general, or not specific enough as an effective vehicle for ASIC simulation.

An ASIC cell (gate) is a complex object which requires care and experience in modeling. But the characteristics of an ASIC cell are such that a few very specific modeling constructs are required, while the full power of VHDL is not. As a result, each attempt to use a general purpose VHDL tool (analyzer and simulator) for sign-off simulation failed because the simulation was too slow and did not offer the required capacity. What follows are some examples of how the specific modeling requirements of an ASIC cell interact with the general nature of VHDL.

3.1. Table Lookup

ASIC cells are often described in terms of a truth table (or a next state table). A table is a very simple and effective format for computation of a value based on an index, the main advantage being the constant access time to the table. In VHDL, such a concept could be represented in several ways, each with its advantages and disadvantages. It is possible to use a table much like the original table in the cell's specifications. But each table (array) access in VHDL requires bounds checking at a cost to performance. A table may also be represented by a case-statement, which requires a conditional branch (a table lookup) and an additional assignment (to get the value), again, at a cost to performance. Or, the table may be represented by an expression which, when computed, returns the desired value. Again, it is easy to see an impact on performance[1].

3.2. Pin-to-Pin Delays

As noted above, an ASIC cell's timing is often specified using a Pin-to-Pin delay model. VHDL, on the other hand, associates delay with a signal assignment, not with a port (pin). To convert between what VHDL can do and how the cell is specified requires an 'if-then-else' code structure to select the appropriate delay value. Again, there is some performance cost.

3.3. Fixed Logic Type

VHDL supports arbitrary user types. ASIC cells require one or two data types at the most (time and logic)[2]. If that specific type (i.e. MVL9) is handled as a regular (user) type and not afforded special treatment, the simulator will suffer in performance and possibly in capacity.

3.4. General Signal Structure

VHDL signals are very powerful. It is possible, given a VHDL signal, to find many properties of such an object. For example, did this signal experience an *event*? Did it have a *transaction*? What was the *last value* of this signal? When was this signal last *active*? In addition, it is possible to derive other signals from a particular given signal (signal attributes). This rich and complex functionality is, to a large extent, not needed in modeling an ASIC design. In fact, the only requirement is to detect an event on a signal (and the transition type). The obvious implication

1. But in some cases an expression is advantages since it will require less space.
2. In some cases an integer type may be required.

is a significant opportunity for memory reduction.

3.5. General Resolution Function

Signals (nets) in an ASIC design are resolved, which means that each can have more than one source. But, unlike general signals in VHDL, they need not have an arbitrary resolution node (resolution function). By knowing in advance what the resolution node is, the simulator can take advantage of optimization opportunities that are unavailable for the general case.

4. Sign-Off In VHDL

As stated above, our motivation was to enhance a VHDL environment that will enable all phases of ASIC design *including* Sign-off. To achieve that goal we had to resolve many of the problems described above. The general approach we used is to agree on a set of VHDL constructs and ASIC library aspects and, while providing a VHDL definition for such required features, to build the essential functionality in to the simulator. The challenge we faced was to do that with minimal loss of flexibility and generality. In the next several sections we describe some detail of our approach.

4.1. General Approach

After collecting and reviewing the requirements we had gathered from some of our ASIC partners, we outlined the following approach:

- Use MVL9 for logic type[1].
- Use parameterized generic primitives. When we could not 'fix' a feature or agree on one specific way this feature should function, we parameterized it. This provided us with maximum flexibility.
- Use table lookup to represent functionality and next-state information. Tables are also used for other portions of the primitives. This made our primitives highly configured. Also as a result, we only required two classes of primitives, one combinational, the other sequential.
- Represent ALL the required functionality in the models, including the specifications of how to select a delay value. Only delay values are annotated.
- Use SDF as well as configuration to do timing back-annotation[2].
- Represent all the required checks in the models, including hazard detection, or timing violation.
- Provide facilities for report generation. We added some generic facilities for reporting of toggle count or bus contention.

As a whole, this approach provided a flexible yet powerful modeling environment. But in addition, we had to provide this environment such that it will allow for high capacity and superior simulation performance. To do that, we built into the kernel as much of the functionality as was possible.

4.2. Built-in Functionality

Instead of executing code generated from VHDL, we pre-coded much of the functionality:

1. But integer is also used in some cases. And in some cases sub-set of MVL9 is used.
2. SDF is the only aspect in our approach that is not VHDL compatible. It is possible however, to create a compatible configuration.

- We builtin the evaluation of components. Since our components are predefined it was possible for us to write the component evaluator in to the simulator. This did not require any loss of functionality or flexibility since the components are parameterized and use generic constants and table lookup to modify the required functionality.
- We builtin a reduced signal data structure. Since the only type we require is MVL9 it was possible to create a greatly reduced signal structure. This improved our capacity and speed.
- We builtin a resolution function. Again, based on the fact that only MVL9 is required, we could build in the resolution function.
- We builtin hazard and violation detection, and the various aspects of reporting and responding to such situations.

Pre-coding as much of the functionality as was possible dramatically reduced our run time memory requirements and greatly enhanced our simulation speed. In addition, many other optimization opportunities existed, of which we did take advantage.

4.3. VHDL Compatibility[1]

It is not practical to show in this paper the complete VHDL description used in the definition of our primitives. To illustrate our VHDL definition we present the VHDL code description of one of the combinational primitives. For simplicity, we have removed some of the generic parameters in the entity interface. In the body of the architecture, we omit many of the checks specified.

```
entity TLU is
   generic (
      N: POSITIVE := 1; -- number of inputs
      TruthTable: val_Z01X_vector;
      TT_Size: integer_vector;
      node_index: integer_vector;
      pin_names: STRING;
      delay_param: delay_table;
      sdt_values: sdt_values_t;
      InMapZ: val_01X_vector;
      OutMapZ: STD_ULOGIC;
      PulseHandling: ph_option := ph_inertial;
      Timing_mesg: BOOLEAN := TRUE;
      Timing_xgen: BOOLEAN := FALSE;
      strn: STRENGTH := strn_X01); -- output strength
   port (
      Input: IN STD_LOGIC_VECTOR (0 TO N-1); -- input high to low
      Output: OUT STD_LOGIC); -- output
end TLU;

-- architecture body --
architecture A of TLU is
   signal current_out : STD_LOGIC := 'U';
   constant pin_index : integer_vector := create_pin_index(N, pin_names);
   alias delay_param_0 : delay_table(0 to delay_param'length-1) is
delay_param;
   alias InMapZ_0 : val_01X_vector(0 to InMapZ'length-1) is InMapZ;
begin -- architecture
```

1. Tonny Yu has developed and written *all* the VHDL code that is used in this section. But since we have mutilated his code so as to fit this paper's format, all errors or typos in the code are the responsibility of the authors of this paper.

```
process (Input)
   variable pend_event : Time := 0 ns;
   variable pend_out : val_ZX := 'X';
   variable final_out : STD_LOGIC;
   variable proj_out : val_ZX := 'X';
   variable proj_delay : Time;
   variable max_delay : Time;
   variable hazard : Boolean;
   variable connect, connect_delayed : val_01X_vector(Input'range) :=
                        (others => 'X');
   variable iInput : INTEGER;

begin
   connect := MAP_INPUT(Input, InMapZ_0); -- connect becomes X01
   if (connect /= connect_delayed) then
      proj_out := TBL_LOOKUP(connect, TruthTable, TT_Size, node_index);
      final_out := tbl_map_output(proj_out, OutMapZ, strn);

 -- timing arcs
      max_delay := 0 ns;
      for i in Input'range loop
         if connect(i) /= connect_delayed(i) then
            proj_delay := DELAY_CALC(pend_out, proj_out,
                    delay_param_0(i)(tran_01),
                    delay_param_0(i)(tran_10),
                    delay_param_0(i)(tran_0Z),
                    delay_param_0(i)(tran_Z1),
                    delay_param_0(i)(tran_1Z),
                    delay_param_0(i)(tran_Z0));
            if (proj_delay > max_delay) then
               max_delay := proj_delay;
               iInput := i;
            end if;
         end if;
      end loop;
   connect_delayed := connect;

   SCHEDULE(final_out, current_out, proj_out, pend_event,
   pend_out, max_delay, hazard, PulseHandling);

   if hazard then
      -- report & schedule Hazard action!
   end if;

   pend_event := proj_delay + Now;
   pend_out := proj_out;
   end if;
end process;

Output <= current_out;
end A;
```

The model above uses a package which defines the function procedures and types used in the model. Again, we only include a short portion from that package, in this case, part the procedure "schedule".

```
procedure SCHEDULE(
   constant input : in STD_LOGIC;
   signal current_out : inout STD_LOGIC;
   constant proj_out : in val_ZX;
   variable pend_event : inout time;
   constant pend_out : in val_ZX;
   variable proj_delay : inout time;
   variable hazard : out Boolean;
   constant PulseHandling : in ph_option) is
begin
   hazard := False;

   pend_event := pend_event - Now;
   if (PulseHandling = PH_INERTIAL) then
      current_out <= input after proj_delay;
   else
      current_out <= transport input after proj_delay;
   end if;

   if (pend_event >= 0 ns) then
      if (pend_out = proj_out) then
         proj_delay := pend_event;
      elsif ((proj_delay < pend_event) or
         (val_ZXtoX01Z(pend_out) /= current_out)) then
      hazard := True;
      end if;
   end if;
end SCHEDULE;
```

4.4. Integration within a system

ASIC design does not rely on simulation alone. Many other tools are used in the design process. Some tools are used to create the library, others are used in manipulation, optimization and generation of a design. Other tools yet, are used to verify the library and the design. We designed our simulation and library strategy to fit with prevailing tools and methodology.

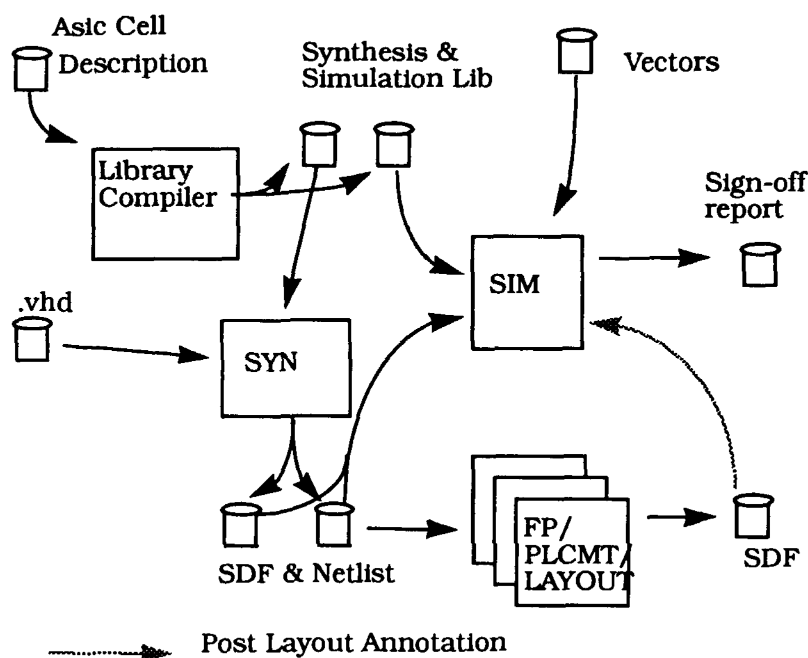

Figure 3. Design Process

By using a library generator (Library Compiler) we enable the use of a common format for description of ASIC cell for Synthesis and Simulation. This ensures compatibility in every design stage. Once the application is verified (using simulation) it is synthesized and targeted to the applicable technology library. It is then simulated again using the simulation library of the corresponding technology. Timing information, from the synthesis delay predictor or from the vendor's delay predictor, is annotated using SDF.

5. Relation to VITAL

VITAL is an industry effort aimed at achieving wide acceptance of VHDL ASIC simulation practices. The main goal of the VITAL program is to enable use of VHDL as a common language for ASIC simulation.

If VITAL is successful, as we hope that it will be, it will become a standard for VHDL ASIC simulation. Taking that into consideration, we have made every effort to ensure that our approach is compatible with VITAL. At this time the available version of VITAL specifications is 2.0, and as a result it is premature to evaluate complete compatibility with VITAL practices.

We define VITAL compatibility as the 'Ability to accept VITAL compliant models and descriptions and produce correct simulation results'. Several aspects of our approach give us reason to be cautiously confident that we will indeed be VITAL compliant.

Both the VITAL approach and our approach are based on **VHDL**. As a result, integrating a VITAL model and one of our models is a simple task which requires no special tools or conversions. This is important since it enables us to create VITAL compliant models simply by **encapsulating** our parameterized **primitives**. To create a VITAL 'primitive' all we would need to do is to instance a specifically parameterized primitive in the architecture of a VITAL entity.

We enjoy a similar advantage in complying with the VITAL timing model, since both our

approach and VITAL's are based on a **pin-to-pin** delay model. The same can also be said about backannotation. VITAL specifies the use of a sub-set of SDF[1]. Our approach uses the same back-annotation format.

6. Conclusions

We have described a solution to ASIC Sign-off in VHDL. Our solution uses VHDL as a definition format and while allowing great flexibility in modeling, attains performance and capacity levels that are otherwise unrealistic. We achieve this by building much of the functionality that is required into the simulation kernel and elaborator. As proof of the concept (but also for practical applications) we provide complete VHDL models, which serve as the definition of our build in functionality. Our solution has been evaluated and adopted by several ASIC vendors and is currently being evaluated by many more.

We have made an effort to ensure that our approach is compatible with VITAL when it becomes applicable.

In summary, we believe our approach is practical and realistic and will be of value to the ASIC design community.

7. References

IEEE Standard VHDL Language Reference Manual, 1987, IEEE Standards Department

Gate Level Engine Simulation, 1993, Synopsys Inc. [Internal Document]

Creating Gate Level Simulation Library, 1993, Synopsys Inc. [Internal Document]

VITAL Model Development Specifications; Version 2.0, 1993, VITAL committee.

8. Acknowledgments

This paper represents the work of many. In Synopsys, the Engineering groups of Simulation and Library Compiler are responsible for the design and implementation of the functionality described in this paper. The Semiconductor Vendor Program personal, also in Synopsys, was very instrumental in getting early requirements from ASIC vendors. Outside of Synopsys, we are thankful to all the ASIC vendors that have contributed (and will continue to contribute) their requirements to us. Participating in the VITAL effort has also been very useful to this effort.

1. At the time this is written, it is unclear just what the VITAL SDF sub-set will be.